Using ATL to define advanced and flexible constraint model transformations

Raphael Chenouard, Laurent Granvilliers, Ricardo Soto

Using ATL to define advanced and flexible constraint model transformations

Raphaël Chenouard[1], Laurent Granvilliers[1], and Ricardo Soto[1,2]

[1] LINA, CNRS, Université de Nantes, France
[2] Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso, Chile
{raphael.chenouard,laurent.granvilliers,ricardo.soto}@univ-nantes.fr

Abstract. Transforming constraint models is an important task in recent constraint programming systems. User-understandable models are defined during the modeling phase but rewriting or tuning them is mandatory to get solving-efficient models. We propose a new architecture allowing to define bridges between any (modeling or solver) languages and to implement model optimizations. This architecture follows a model-driven approach where the constraint modeling process is seen as a set of model transformations. Among others, an interesting feature is the definition of transformations as concept-oriented rules, i.e. based on types of model elements where the types are organized into a hierarchy called a metamodel.

1 Introduction

Constraint programming (CP) systems must combine a modeling language and a solving engine. The modeling language is used to represent problems with variables, constraints, or statements. The solving engine computes assignments of variables satisfying the constraints by exploring and pruning the space of potential solutions. This paper considers the constraint modeling process as constraint model transformations between arbitrary modeling or solver languages. It follows several important consequences on the architecture of systems and user practices.

Constraint programming languages are rich, combining common constraint domains, e.g. integer constraints or linear real constraints, with global constraints like `alldifferent`, and even statements like `if-then-else` or `forall`. Moreover the spectrum of syntaxes is large, ranging from computer programming languages like Java or Prolog to high-level languages intended to be more human-comprehensible. This may be contrasted with the existence of a standard language in the field of mathematical programming, which improves model sharing, writing and understanding. The quest of a standard CP language is a recent thread, dating back to the talk of Puget [15]. Another important concern is to employ the best solving technology for a given model. As a consequence, a new kind of architecture emerged. The key idea is to map models written with a high-level CP language to many solvers. For instance within the G12 project,

MiniZinc [13] is intended to be a standard modeling language, and Cadmium [3] is able to map MiniZinc models to a set of solvers. Essence [5] is another CP platform offering an high level modeling language refining Essence specifications to Essence' models using Conjure [6]. Then hand-written translators can generate models for several different solvers. The role of a mapping tool is to bridge modeling and solver languages and to optimize models for improving the solving process. Cadmium is based on Constraint Handling Rules [8] and is the the closest CP platform from our model-driven approach.

In our approach, we suppose that any CP language can be chosen at the modeling phase. In fact, finding a standard language is hard and existing languages have their own features. It then becomes necessary to define mappings between any (pure modeling or solver) languages. This is just the first goal of the new architecture for constraint model transformations defined in the sequel. It follows many advantages:

- Any user may choose its favourite modeling language and the known best solving technology for a given problem provided that the transformation between languages is implemented.
- It may be easy to create a collection of benchmarks for a given language from different source languages. This feature may speed up prototyping of one solver, avoiding hand rewriting of problems into the solver language.
- A given problem may be handled using different solving technologies. Users may not have to play with solver languages.

To this end, we define a generic and flexible pivot model (i.e. an intermediate model) to which any language is mapped. Considering a new language in this framework only requires a parser and a generally simple transformation to the pivot model.

The second goal is to define refactoring operations and optimizations of constraint models using declarative rules. Implementing them over pivot models guarantees the independence from external languages. In other words every operation is implemented once, by means of a so-called concept-oriented rule. In our model engineering approach the elements of models are specified within metamodels, which can be seen as a hierarchy of concepts or types. The rules are able to filter models according to these types, which may be more powerful than syntax-oriented rules.

The third goal is to apply the best transformations for given solving technologies. For instance, a matrix with a few non null elements could be transformed into a sparse matrix when using a linear algebra package. The selection of transformation steps is implemented as a sequential procedure, applying transformations until at least pivot models fit the structure requirements of the target language.

This architecture has been fully implemented using a model-driven engineering (MDE) approach [14]. MDE tools enable us to separate the grammar concerns from modeling concepts using dedicated tools and languages like TCS [11] and ATL [12,10]. The main advantage is that we can reason about concepts and

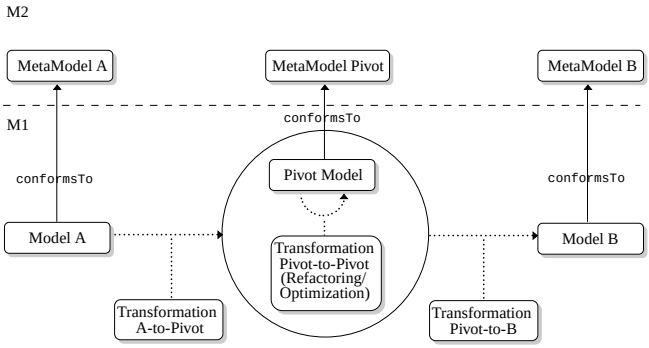

Fig. 1. General transformation framework.

their relations through a metamodel. Transformations are specified by defining matchings between concepts at the metamodel level of abstraction. Thus, grammar concerns are relegated into the foreground, while concepts processing becomes the major task.

With respect to previous works, e.g. [4], the new architecture gives more freedom in constraint modeling. s-COMMA is not always the source modeling language and refactoring steps can be chosen. Thus, users can play with any modeling language, until it is mapped to our platform. Dealing with a solver does not require to manipulate its language. Moreover, handling a new language or a new transformation in the system requires a few work. The main limitation of our approach is that only the modeling fragments of languages can be processed i.e., the declarative part. It is not possible to partially execute a computer program that builds the constraint store.

This paper is organized as follows. Section 2 presents an overview of our general transformation framework. Next section introduces the metamodels of two CP languages illustrated on a well-known problem. The pivot metamodel and the transformation rules are introduced in Section 4. Section 5 presents the whole model-driven process including the possibility of selecting relevant mappings. The related work and a conclusion follow.

2 The Model-Driven Transformation Framework

Figure 1 depicts the architecture of our model-driven transformation framework, which is classically divided in two layers M1 and M2 [14]. M1 holds the models representing constraint problems and M2 defines the semantic of M1 through metamodels. Metamodels describe the concepts appearing in models, e.g. constraint, variable, or domain, and the relations among these concepts, e.g. inheritance, composition, or association. In this framework, transformation rules are defined to perform a complete translation in three main steps: translation from source model A to the pivot model, refactoring/optimization on the pivot model, and translation from the pivot model to target model B. Models A and B may

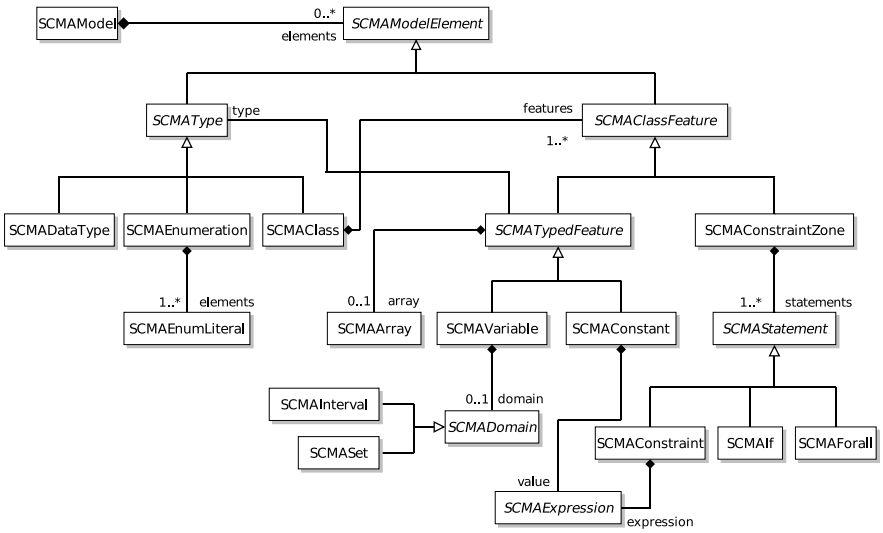

Fig. 2. Extract of the s-COMMA metamodel.

be defined through any CP languages. The pivot model may be refined several times in order to adapt it to the desired target model (see Section 4).

A main feature resulting from a model-driven engineering approach is that transformation rules operate on the metamodel concepts. For instance, unrolling a `forall` loop is implemented once over the `forall` concept, which is independent from the many syntaxes of `forall` in CP languages. In fact, no grammar specification is required for the pivot model. Syntax specifications of CP languages must be defined separately using specific tools achieving text-to-model or model-to-text mappings like TCS [11], which implement both tasks.

3 A Motivating Example

In this section, we consider two CP languages, and we motivate the needs and the means for implementing transformations between them.

ECL^{i}PS^e [17] is chosen as a leading constraint logic programming system. s-COMMA [16] is an object-oriented constraint language developed in our team. Their metamodels are partially depicted in Figure 2 and 3 using UML class diagram notation. The roots of these hierarchies are equivalent, such that the model concept represents the complete constraint problem to be processed.

In s-COMMA, a model is composed of a collection of model elements. A model element is either an enumeration, or a class, or a constant. Each class is composed of a set of class features which can be specialized in variables, constant or constraint zones. Variable with a type defined as a class is an object. Constraint zones are used to group constraints and other statements such as conditionals and loops. The concepts of global constraints and optimization objective are not

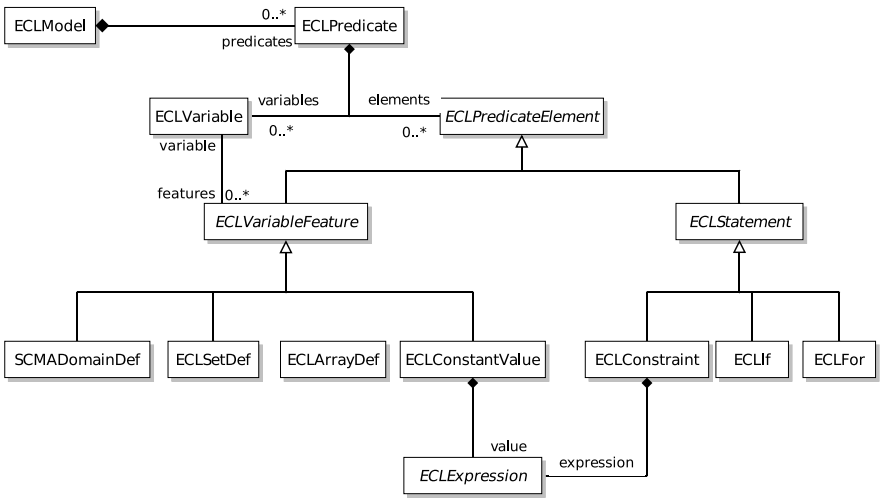

Fig. 3. Extract of the ECL^{i}PS^e metamodel.

shown here, but can be also defined. The concept of expressions are not detailed in this paper since it is based on classical operatored expressions using boolean, set and arithmetic operators.

In the ECL^{i}PS^e metamodel, we propose to define a model as a collection of predicates holding predicate elements and variables. Predicate elements are variable features or statements. Variables features is either a constant value assignment, a domain definition, an array or a set definition related to a variable. In fact, we consider that variables are implicitly declared through their features.

Considering the well-known problem of the social golfers, Figure 4 and 5 show two versions of the same problem using s-COMMA and ECL^{i}PS^e languages. This problem considers a group of $n = g \times s$ golfers that wish to play golf each week, arranged into g groups of s golfers, the problem is to find a playing schedule for w weeks such that no two golfers play together more than once.

The s-COMMA model is divided in a data file and a model file. The data file contains the golfer names encoded as an `Enum` concept at line 1 and the problem dimensions defined by means of constants (size of groups, number of weeks, and groups per week). The model file represents the generic social golfers problem using the `Model` concept. The problem structure is captured by the three classes `SocialGolfers`, `Group`, and `Week`, which are conformed to the `Class` concept. The `Group` class owns the `players` attribute corresponding to a set of golfers playing together, each golfer being identified by a name given in the enumeration from the data file. In this class, the constraint zone `groupSize` (lines 30 to 32) restricts the size of the golfers group. The `Week` class has an array of `Group` objects and the constraint zone `playOncePerWeek` ensures that each golfer takes part of a unique group per week. Finally, the `SocialGolfers` class has an array of `Week` objects and the constraint zone `differentGroups` states that each golfer never plays two times with the same golfer throughout the considered weeks.

```
1  // Data file
2  enum Name := {a,b,c,d,e,f,
       g,h,i};
3  int s := 3; //size of
       groups
4  int w := 4; //number of
       weeks
5  int g := 3; //groups per
       week
6
7  // Model file
8  main class SocialGolfers {
9   Week weeks[w];
10  constraint
        differentGroups {
11   forall(w1 in 1..w) {
12    forall(w2 in w1+1..w) {
13     card(weeks[w1].groups[
           g1].players
           intersect weeks[w2
           ].groups[g2].
           players)<= 1;
14    }
15   }
16  }
17 }
18 class Week {
19  Group groups[g];
20  constraint
        playOncePerWeek {
21   forall(g1 in 1..g) {
22    forall(g2 in g1+1..g) {
23     card(groups[g1].
           players intersect
           groups[g2].players
           ) = 0;
24    }
25   }
26  }
27 }
28 class Group {
29  Name set players;
30  constraint groupSize {
31   card(players) = s;
32  }
33 }
```

Fig. 4. The social golfers problem expressed in s-COMMA.

```
1  socialGolfers(L):-
2   S $= 3,
3   W $= 4,
4   G $= 3,
5   intsets(
        WEEKS_GROUPS_PLAYERS
        ,12,1,9),
6   L = WEEKS_GROUPS_PLAYERS,
7
8   (for(W1,1,W), param(L,W,G
        ) do
9    (for(W2,W1+1,W), param(L
         ,G,W1) do
10    (for(G1,1,G), param(L,G
          ,W1,W2) do
11     (for(G2,1,G), param(L,
           G,W1,W2,G1) do
12      V1 is G*(W1-1)+G1,
13      nth(V2,V1,L),
14      V3 is G*(W2-1)+G2,
15      nth(V4,V3,L),
16      #(V2 /\ V4, V5),V5
           $=< 1
17     )
18    )
19   )
20  ),
21  (for(WEEKS,1,W),param(L,G
       ) do
22   (for(GROUPS,1,G), param(
        L,S,W,G,WEEKS) do
23    V6 is G*(WEEKS-1)+
         GROUPS,
24    nth(V7,V6,L),
25    #(V7, V8), V8 $= S,
26
27    (for(G1,1,G),param(L,G,
         WEEKS) do
28     (for(G2,G1+1,G),param(
          L,G,WEEKS,G1) do
29      V9 is G*(WEEKS-1)+G1,
30      nth(V10,V9,L),
31      V11 is G*(WEEKS-1)+G2
           ,
32      nth(V12,V11,L),
33      #(V10 /\ V12, 0)
34     )
35    )
36   )
37  ),
38
39  label_sets(L).
```

Fig. 5. The social golfers problem expressed in ECL^{i}PS^e.

Figure 5 depicts the ECL^{i}PS^e model resulting from an automatic transformation of the previous s-COMMA model. The problem is now encoded as a single predicate whose body is a sequence of atoms. The sequence is made of the problem dimensions, the list of constrained variables `L`, and three statements resulting from the transformation of the three s-COMMA classes. It turns out that parts of both models are similar. This is due to the sharing of concepts in the underlying metamodels, for instance constants, `forall` statements, or constraints. However, the syntaxes are different and specific processing may be required. For instance, the `forall` statement of ECL^{i}PS^e needs the `param` keyword to declare parameters defined outside of the current scope, e.g. the number of groups `G`.

The treatment of objects is more subtle since they must not participate to ECL^{i}PS^e models. Many mapping strategies may be devised, for instance mapping objects to predicates [16]. Another mapping strategy is used here, which consists in removing the object-based problem structure. Flattening the problem requires visiting the many classes through their inheritance and composition relations. A few problems to be handled are described as follows. Important changes on the attributes may be noticed. For example, the `weeks` array of `Week` objects defined at line 9 in Figure 4 is refactored and transformed to the `WEEKS_GROUPS_PLAYERS` flat list stated at line 5 in Figure 5. It may be possible to insert new loops in order to traverse arrays of objects and to post the whole set of constraints. For instance, the last block of for loops in the ECL^{i}PS^e model (lines 27 to 39) has been built from the `playOncePerWeek` constraint zone of the s-COMMA model, but there is two additional for loops (lines 21 and 22) since the `Week` instances are contained in the `weeks` array. Another issue is related to lists that cannot be accessed in the same way than arrays in s-COMMA. Thus, local variables (`V`$_i$) and the well-known `nth` Prolog built-in function are introduced in the ECL^{i}PS^e model.

4 Pivot metamodel and refactoring rules

The pivot model of a constraint problem is an intermediate model to be transformed by rules. The rules may be chained to implement complex transformations. In the following, the pivot and some structural refactoring and optimization rules are presented.

4.1 Pivot metamodel

Our pivot model has been designed to support as much as possible the constructs present in CP languages, for instance variables of many types, data structures such as arrays, record, classes, first-order constraints, common global constraints, and control statements. We believe that it is better and simpler to establish a general CP metamodel, while it is more complex to find a standard CP concrete syntax.

Figure 6 depicts the metamodel associated to pivot models. A pivot model is composed of a collection of elements, divided in three main concepts: types,

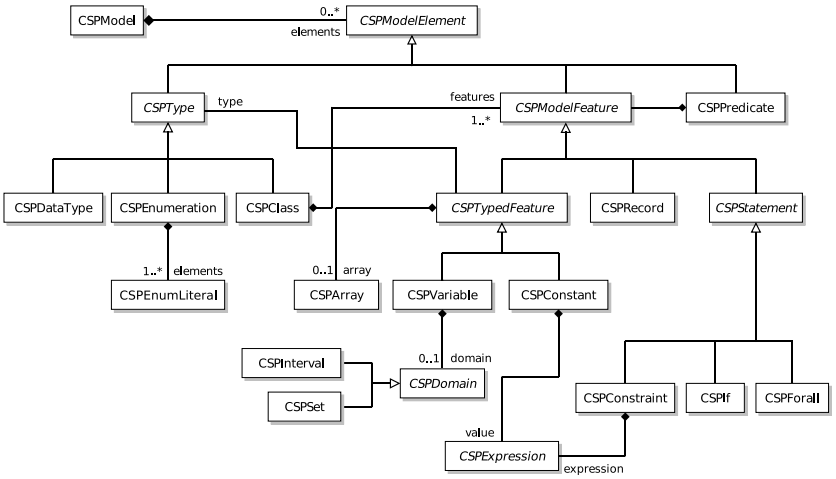

Fig. 6. Extract of the pivot metamodel.

features and the concrete concept of predicate. The inheritance tree of types is the same as in the s-COMMA metamodel (see Figure 2). The inheritance tree for model features is also quite similar, except for the concept of record which is an untyped collection of features.

4.2 Pivot model refactoring

We define several refactoring steps on pivot models in order to reduce the possible gap between source and target model. These steps are implemented in several model transformations, most of them being independent from the others. The idea is to refine and optimize models in order to fit the target languages supported concepts.

Model transformations are implemented in the declarative transformation rule language ATL [12]. This rule language is based on a typed description of models to be processed, namely their metamodel. In this way, rules are able to clearly state how concepts from source metamodels are mapped to concepts from the target ones. For the sake of simplicity, only a few of the more representative rules of transformations are shown. ATL helpers are not detailed, but they only consist of OCL navigation.

Composition flattening This refactoring step replaces object variables by duplicating elements defined in their class definition. Names of duplicated variables are prefixed using their container name in order to avoid naming ambiguities. This refactoring step processes object variables and their occurrences, while other entities are copied without modification. In fact, two ATL transformations are defined to ease each refactoring step. The first one removes classes and object variables by replacing them by the concept of record (see Figure 7).

It can be highlighted that there is no ATL rule where the source pattern matches elements being instances of `CSPClass`. Thus, they are implicitly removed from models (obviously no rule creates class instances). The second transformation removes records to get flattened variables (see Figure 8).

```
1  rule Model {
2    from
3      s : Pivot!CSPModel
4    to
5      t : Pivot!CSPModel (
6        name <- s.name,
7        elements <- s.elements
8  }
9  rule Variable {
10   from
11     s : PivotCSP!CSPVariable (
12       not s.mustBeDuplicated
13     )
14     to
15       t : PivotCSP!CSPVariable (
16         name <- s.name,
17         type <- s.type,
18         domain <- s.domain,
19         isSet <- s.isSet,
20         array <- s.array
21       )
22 }
23 rule Variable2Record {
24   from
25     s : PivotCSP!CSPVariable (
26       s.isObject
27     )
28     to
29       t : PivotCSP!CSPRecord (
30         name <- s.name,
31         array <- s.array,
32         elements <- s.type.features->collect(f|
33           thisModule.duplicate(f)
34         )
35       )
36 }
```

Fig. 7. An extract of ATL rules used to remove the concept of class in pivot models.

In Figure 7, the first rule (lines 1 to 8) is used to copy the root concept of model. Most of other concepts are duplicated with similar rules like the the second one (lines 9 to 22). The helper `mustBeDuplicated` is defined for each `CSPModelFeature` and it returns true when: (1) the considered element is an object variable (its type is a class) or (2) it is a feature of a class. Using the last rule, object variables are replaced by records. The helper `isObject` returns true only if the type of variables is a class. In this rule, features of variable classes are browsed using OCL navigation (`collect` statement over `s.type.features`). The rule `duplicate` is applied on each feature. This rule is lazy and abstract. It

is specialized for each `CSPModelFeature` concrete sub-concepts and it creates as many features as it is called.

The second transformation processes records by replacing them by their set of elements. This is easily done by collecting their elements from their container as shown on Figure 8 at lines 7 to 11. The helper `getAllElements` returns the set of `CSPModelFeature` within a record or a hierarchy of records.

```
1  rule CSPModel {
2    from
3      s : PivotCSP!CSPModel
4    to
5      t : PivotCSP!CSPModel (
6        name <- s.name,
7        elements <- s.elements->union(s.elements->select(r|
8            r.oclIsTypeOf(PivotCSP!CSPRecord)
9          )->collect(r|
10           r.getAllElements
11         )->flatten())
12     )
13 }
14 rule RecordArray {
15   from
16     s : PivotCSP!CSPRecord (
17       (not s.array.oclIsUndefined()) and
18       s.elements->select(e|
19         e.oclIsKindOf(PivotCSP!CSPStatement)
20       )->size()>0
21     )
22     to
23       t : PivotCSP!CSPForall (
24         index <- i,
25         constraints <- s.elements->reject(e|
26             e.oclIsKindOf(PivotCSP!CSPTypedElement)
27           ),
28         i : PivotCSP!CSPIndexVariable (
29           name <- s.name,
30           domain <- d
31         ),
32         d : PivotCSP!CSPIntervalDomain(
33           lower <- l,
34           upper <- thisModule.duplicateExpr(s.array.n)
35         ),
36         l : PivotCSP!CSPIntVal(
37           value <- 1
38         )
39 }
```

Fig. 8. Main ATL rules used to remove the concept of record in pivot models.

However, some other complex rules must be defined to process arrays of records, (formerly arrays of object variables). Indeed, contained statements have to be encapsulated in a for loop to take into account the constraints for all objects in the array. This task is performed by the rule `RecordArray` which create a new for loop over the record statements (lines 25 to 27). A new for loop requires also a new index variables with its domain (lines 28 to 38).

Using the concrete syntax of s-COMMA, Figure 9 shows the result of this refactoring step. The name of the variable at line 1 corresponds to the concatenation of all object variable names. The two for loops (lines 2 and 3) were created from the arrays of objects using their name for index variables.

```
1    int set weeks_groups_players[w*g] in[1, 9],
2    forall weeks in [1,w] {
3      forall groups in [1,g] {
4        card(weeks_groups_players[weeks*w+groups])= g,
5        ...
6      }
7    }
```

Fig. 9. Extract of the social golfers pivot model after composition removal and enumeration removal transformations.

Enumeration removal During this refactoring step, enumeration variables are replaced by integer variables with a domain defined as an interval from one to the number of elements within the enumeration. Line 1 in Figure 9 shows the result of this transformation on the enumeration called `Name` in the social golfers model: the variable has an integer domain from 1 to 9 replacing the set of nine values $\{a, b, c, d, e, f, g, h, i\}$. In the same way, occurences of `CSPEnumLiteral` are replaced by their position in the sequence of elements of the enumeration type.

Other implemented refactoring steps Some other generic refactoring steps have been implemented in ATL to handle some structural needs. They are not detailed since their complexity is similar to the previous examples and to detail all of them is not the scope of this paper.

- If statements can be replaced by one constraint based on one or two boolean implications. For instance, $if\ a\ then\ b\ else\ c$ becomes $(a \rightarrow b) \wedge (\neg a \rightarrow c)$.
- Loop structures can be unrolled, i.e. the loop is replaced by the whole set of constraints it implicitly contains. Within expressions, the iterator variable used by the loop structure is replaced by an integer corresponding to the current number of loop turns.
- Expressions can be simplified if they are constants. Boolean and integer expressions are replaced by their evaluation. Real expressions are not processed, because of real number rounding errors. More subtle simplifactions can be performed on boolean expressions such as $a \vee \neg a$ that is always true. Only atomic boolean elements are processed by this last step.
- Matrices are not allowed in all CP language, thus they can be replaced by one dimension arrays. Their occurrences in expressions must also be adapted: the index of the array is computed as follows: $m[i, j]$ becomes $m[j + (i * ncols)]$, where $ncols$ is the number of columns of the matrix m.

– The ECL^{i}PS^e language does not allow some sort of expressions. For instance, arrays of int sets cannot be accessed like other arrays with `'[ ]'`. Thus, an ECL^{i}PS^e specific transformation processes expressions and introduces local variables if needed, as shown on Figure 5 with $\mathtt{V}_i$ variables and `nth` predicate calls.

5 Handling CP languages and transformation chains

In this section, we describe the whole transformation chain from a given CP language to another language.

5.1 Parsing CP languages

The front-end of our system parses a source CP language file to get a model representation (on which transformation rules act) matching the concepts of the CP language (injection phase). The back-end generates the code in the target CP language (extraction phase) from the model representation. Interfacing CP languages and metamodels is implemented by means of the TCS tool [11]. This tool allows one to smoothly associate grammars and metamodels. It is responsible for generating parsers of CP languages and also code generators.

Figure 10 depicts an extract of the TCS file for s-COMMA. In a TCS file every concrete concept must have a corresponding *template* to be matched. For instance, the `SCMAClass` template implements the grammar pattern for class declarations using at the same time features of this concept defined in the metamodel of s-COMMA. At parsing time on the s-COMMA social golfers example (see Figure 4, the `"class"` token is matched for the week class statement. Then `Week` is processed as the `name` attribute (a string in the metamodel) of a new class instance. Then the `"{"` token is recognized and the class features (the array of groups and the constraint) are processed by implicit matchings to their corresponding templates using the `features` reference. Finally the `"}"` token terminates the pattern description. In the `SCMAClass` template (lines 4 to 8), several TCS keywords are used. Here is a description of the most important keywords use in Figure 10:

– *context* defines a local symbol table.
– *addToContext* adds instances to the current symbol table.
– *refersTo* accesses to the symbol tables according to the given parameter (here the name) to check the existence of an already declared element.

5.2 Model checking rules

The presented metamodels (see section 2) and the previous subsection show how to get CP language models. However, many irrelevant or erroneous models can be obtained without any additional checking [2]. For instance, variables may

```
1   template SCMAModel main context
2     : elements;
3
4   template SCMAClass context addToContext
5     : (isMain ? "abstractmain") (isAbstract ? "abstract") "
          class" name
6         (isDefined(superTypes) ? "extends"
7           superTypes{separator=",",refersTo=name,
                importContext})
8    "{" [ features {separator=","} ] "}" ;
9
10  template SCMAVariable addToContext
11    : (isSet ? "set" : "") type {refersTo=name}
12      name (isDefined(array) ? array)
13        (isDefined(domain) ? "in" domain);
```

Fig. 10. Linking the grammar and the metamodel of s-COMMA with TCS.

be defined with empty domains or expressions may be ill made (e.g. several equalities in an equality constraint).

Several ATL transformations are used to check source models. We transform a source CP model to a model conform to the metamodel Problem defined in the ATL zoo[3]. A `Problem` model corresponds to a set of `Problem` elements. This concept is only composed of three features:

- `severity` is an attribute with an enumerated type which possible values are: error, warning and critic.
- `location` is a string used to store le location of the problem in the source file.
- `description` is a string used to defined a relevant message to descibe the problem.

Multiple ATL rules have been implemented to check models. Here is an extract of the list of properties to check:

- Some type checking on expressions. Operands must have a consistent type with the operator. For instance, an equality operator may operate on arithmetic expressions.
- The consistency of variable domains : they must be based on constant expressions and interval domains must have a lower bound smaller than the upper bound.
- No composition or inheritance loops in s-COMMA.

5.3 Chaining model transformations

After the injection step or before the extraction step, models have to be transformed with respect to our pivot metamodel. All the refactoring steps presented in Section 4.2 are clearly not necessary in a transformation chain. Indeed, it

[3] `http://www.eclipse.org/m2m/atl/atlTransformations/#KM32Problem`

clearly depends on the modeling structures of the source and target CP languages. The idea is to use most of constructs supported by the target language to have a target model close, in terms of constructs, to our source model. For instance, when translating a s-COMMA model to ECL^{i}PS^e, we should transform the objects. So, we choose the composition flattening step. We also need the enumeration removal and other refactoring steps such as the use of local variables and `nth` predicates. Optionally, we may select the expression simplification steps.

The whole transformation chain is based on three kind of tasks: (1) injection/extraction steps, (2) transformation steps from/to the pivot model, (3) relevant refactoring steps. Transformation chains are currently performed using Ant scripts[4]. These scripts are hand-written, but they can be automatically generated using the am3 tool [1] and the concept of megamodel [7] to get a graphical interfaces to manage terminal models, metamodels and complex transformation chains. However, Automating the building of transformation chains is not possible with current tools. It would require to deeply analyze models and transformations to build relevant transformation chains.

6 Experiments

The benchmarking study was performed on a 2.66Ghz computer with 2GB RAM running Ubuntu. The ATL regular VM is used for all model-to-model transformations, whereas TCS achieve the text-to-model and model-to-text tasks. Five CP problems were used to validate our approach as shown in Table 1. The second column represents the number of lines of the s-COMMA source files. The next columns correspond to the time of atomic steps (in seconds): model injection (Inject), transformations from s-COMMA to Pivot (s-to-P), refactoring composition structures (Comp), refactoring enumeration structures (Enum), transformations from Pivot to ECL^{i}PS^e (P-to-E), and target file extraction (Extract). The next column details the total time of complete transformation chains, and the last column corresponds to the number of lines of the generated ECL^{i}PS^e files.

Problems	Lines (-)	Inject (s)	s-to-P (s)	Comp (s)	Enum (s)	P-to-E (s)	Extract (s)	Total (s)	Lines (-)
SocialGolfers	42	0.107	0.169	0.340	0.080	0.025	0.050	0.771	38
Engine	112	0.106	0.186	0.641	0.146	0.031	0.056	1.166	78
Send	16	0.129	0.160	0.273	-	0.021	0.068	0.651	21
StableMarriage	46	0.128	0.202	0.469	0.085	0.027	0.040	0.951	26
10-Queens	14	0.132	0.147	0.252	-	0.017	0.016	0.564	12

Table 1. Times for complete transformation chains of several classical problems.

The transformation chain is efficient for these small problems. The text file injection and extraction are fast. The parsing phase is more expensive than the

[4] http://wiki.eclipse.org/index.php/AM3_Ant_Tasks

extraction, since it requires the management of symbol tables. The extraction phase settle for reading the ECL^{i}PS^e model. It can also be noticed that model transformations to and from the pivot are quite efficient, more especially the transformation to ECL^{i}PS^e model. It can be explained by the refactoring phases on the pivot model which simplify and reduce the data to process. We see that the composition flattening step is the more expensive. In particular, the Engine problem exhibits the slowest running time, since it corresponds to the design of an engine with more object compositions.

Problems	Inject (s)	s-to-P (s)	Comp (s)	Forall (s)	P-to-E (s)	Extract (s)	Total (s)	Lines (-)	Total/Lines (-)
5-Queens	0.132	0.147	0.252	0.503	0.071	0.019	1.124	80	≈0.014
10-Queens	0.132	0.147	0.252	1.576	0.280	0.060	2.447	305	≈0.008
15-Queens	0.132	0.147	0.252	3.404	0.659	0.110	4.704	680	≈0.007
20-Queens	0.132	0.147	0.252	6.274	1.224	0.178	8.207	1205	≈0.006
50-Queens	0.132	0.147	0.252	32.815	13.712	1.108	48.166	7505	≈0.006
75-Queens	0.132	0.147	0.252	80.504	54.286	2.456	137.777	16880	≈0.008
100-Queens	0.132	0.147	0.252	175.487	126.607	4.404	307.029	30005	≈0.010

Table 2. Time of complete transformation chains of the N-Queens problem.

Table 2 presents seven different sizes of the N-Queens problem where the loop unrolling step has been applied. This experiment allows us to check the scalability of our approach according to model sizes. It can be analyzed through the ratio given in the last column which aims at quantifying the efficiency of a transformation chain considering the execution time per generated lines.

As shown on this table, the ratio first decreases, but after 50-Queens it slowly grows up. In fact, the first four row ratios are impacted by the steps before the loop unrolling process, but for the last three rows they become neglectible comparing to the whole execution time. It may be noticed that for big problems (after 50-Queens) the ratio smoothly increases. We can thus conclude that our approach is applicable even for huge models, although translations times are not the major concerns in CP.

7 Conclusion and Future Work

In this paper, we propose a new framework for constraint model transformations. This framework is supported by a set of MDE tools that allow an easy design of translators to be used in the whole transformation chain. This chain is composed by three main steps: from the source to the pivot model, refining of the pivot model and from the pivot model to the target. The hard transformation work (refactoring/optimization) is always performed by the pivot which provide reusable and flexible transformations. The transformations from/to pivot become simple, thus facilitating the integration of new language transformations. In this paper, only two languages are presented, but translation processes with Gecode and Realpaver [9] are already implemented.

In a near future, we intend to increase the number of CP languages our approach supports. We also want to define more pivot refactoring transformations to optimize and restructure models. Another major outline for future work is to improve the management of complex CP models transformation chains. Models can be qualified to determine their level of structure and to automatically choose the required refactoring steps according to the target language.

References

1. M. Barbero, F. Jouault, and L. Bézivin. Model driven management of complex systems: Impementing the macroscope's vision. In *15th International Conference on Engineering of Computer-Based Systems*, 2008.
2. J. Bézivin and F. Jouault. Using atl for checking models. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT)*, Tallinn, Estonia, 2005.
3. S. Brand, G. J. Duck, J. Puchinger, and P. Stuckey. Flexible, rule-based constraint model linearisation. In P. Hudak and D. Warren, editors, *Practical Aspects of Declarative Languages*, volume 4902 of *LNCS*, pages 68–83. Springer, 2008.
4. R. Chenouard, L. Granvilliers, and R. Soto. Model-driven constraint programming. In *ACM SIGPLAN PPDP*, pages 236–246, Valencia, Spain, 2008.
5. A. M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of essence: A constraint language for specifying combinatorial problems. In *IJCAI*, pages 80–87, 2007.
6. A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The Rules of Constraint Modelling. In *IJCAI 2005*, pages 109–116, Edinburgh, Scotland, 2005.
7. M. Fritzsche, H. Bruneliere, B. Vanhooff, Y. Berbers, F. Jouault, and W. Gilani. Applying megamodelling to model-driven performance engineering. In *16th Annual IEEE ECBS*, San Fransisco, USA. April 13-16, 2009.
8. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, June 2009. to appear.
9. L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.
10. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).
11. F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Conference on Generative Programming and Component Engineering (GPCE 2006)*, pages 249–254, 2006.
12. F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events 2005*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.
13. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In C. Bessière, editor, *13th International CP Conference*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
14. OMG. *Model Driven Architecture (MDA) Guide V1.0.1, 2003*. http://www.omg.org/cgi-bin/doc?omg/03-06-01.
15. J.F. Puget. Constraint programming next challenge: Simplicity of use. In *CP 2004*, LNCS 3258, pages 5–8, 2004.

16. R. Soto and L. Granvilliers. The design of comma: An extensible framework for mapping constrained objects to native solver models. In *IEEE ICTAI 2007*, pages 243–250, 2007.
17. M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming, 1997.